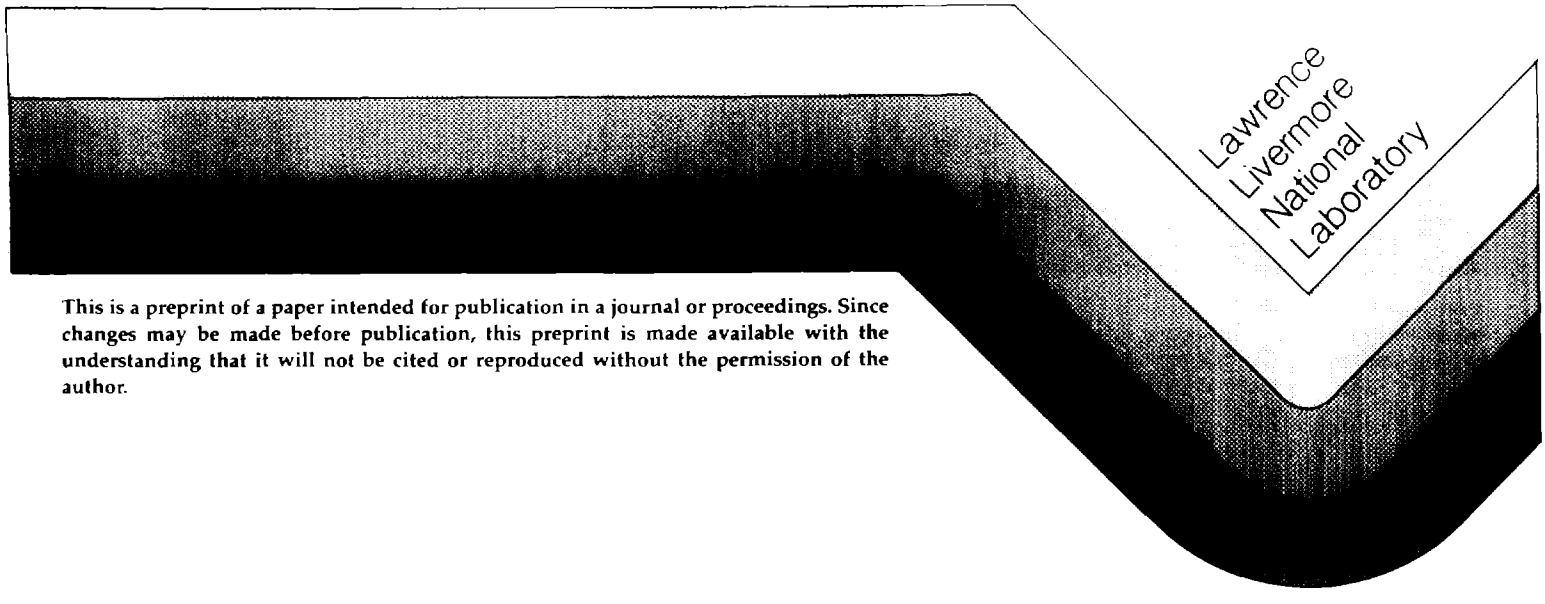
UCRL- 95075
PREPRINT

DESIRE CHARACTERISTICS OF A GENERIC 'NO FRILLS' SOFTWARE ENGINEERING TOOLS PACKAGE

John J. Rhodes

THIS PAPERS WAS PREPARED FOR SUBMITTAL TO
STRUCTURED DEVELOPMENT FORUM VIII
SEATTLE, WASHINGTON
AUGUST 11-15, 1986

JULY 29, 1986



Note

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Disclaimer

Desired Characteristics of a Generic 'No Frills' Software Engineering Tools Package

John J. Rhodes

Applications Systems Division
Computations Department
Lawrence Livermore National Laboratory

Abstract

Increasing numbers of vendors are developing software engineering tools to meet the demands of increasingly complex software systems, higher reliability goals for software products, higher programming labor costs, and management's desire to more closely associate software lifecycle costs with the estimated development schedule. Some vendors have chosen a dedicated workstation approach to achieve high user interactivity through windowing and mousing. Other vendors are using multi-user mainframes with low cost terminals to economize on the costs of the hardware and the tools software. For all of the potential customers of software tools, the question remains: *What are the minimum functional requirements that a software engineering tools package must have in order to be considered useful throughout the entire software lifecycle?* This paper describes the desired characteristics of a non-existent but realistic 'no frills' software enginering tools package.

This paper is based upon the iterative process model of software development shown in Figure 1. The development methodology in most popular use at LLNL is the top-down structured analysis and design approach ala DeMarco [1] and Yourdon/Constantine [2]. This methodology supports the separation of software development into stages each of which has a definable product which can be reviewed, corrected, and approved before proceeding to the next step. The outputs of this methodology are sometimes augmented by an information model [3]) and it is this methodology for which a number of automated tools are being introduced into the marketplace.

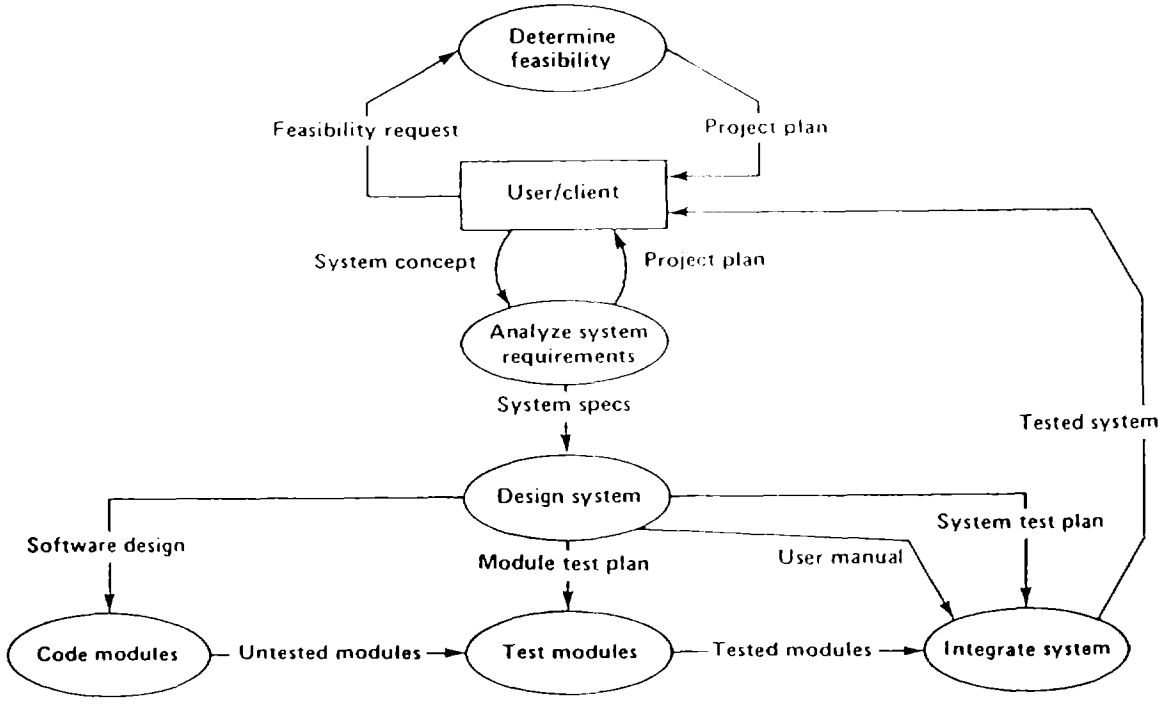


Figure 1 A Process Model for Software Development

supports the separation of software development into distinct stages, each of which has a definable product which can be reviewed, corrected, and approved before proceeding to the next step. This top-down development methodology is sometimes augmented by an information model [3] and there exist various extensions to support real-time software systems. It is this combined methodology for which a number of automated tools are being introduced into the marketplace. One way to determine the desired characteristics of these tools is to examine the product generated at each step of the development process and explore how its production can be automated. For the purpose of this discussion, we will assume that the software development process consists of 6 steps, each of which has a well-defined software-related output, as shown in Table 1 below.

Input	Process	Product
Request	Analyze Feasibility	Feasibility Analysis
Concept	Analyze Requirements	System Specifications
System Specs	Design System	System Design
Software Design	Code Modules	Untested Modules
Module Test Plan, Untested Modules	Test Modules	Tested Modules
System Test Plan, User Manual, Tested Modules	Integrate Modules	Integrated, Tested System

Table 1 Software Development Products

Feasibility Analysis

The output of the feasibility analysis is producer-dependent and may take a number of different forms, depending on the proposed product. It may include market surveys, resource evaluations, and technical feasibility analyses. Many times a feasibility analysis is related to incremental improvements or modifications to existing products or competitors' products. The analysis may involve the examination of differences in operating environments, such as would be the case in porting a software package from one computer system to another or it may involve the potential application of a new technology to an existing problem. Automation assistance of the process of generating feasibility analyses when the process itself is so variable and environment-dependent is difficult to quantify in a general sense; however, advanced document processing tools with such capabilities as easy-to-create embedded drawings (particularly those types of drawings that are frequently found in software proposals), multi-user access, automatic paragraph and page numbering, and automatic index generation can prove very useful. The utilization of easy-to-use prototyping tools (to allow fast prototyping of user interfaces or of critical algorithms) can prove valuable in testing a concept prior to establishing system requirements. Since a part of the feasibility analysis is a preliminary project plan, a complete tools package should include project plan document templates which can be accessed by the advanced document processing system.

Requirements Analysis

The output of the requirements analysis is a system specification document. This document consists of a *system survey*, which describes the existing system (if it exists) and identifies differences between the existing system and

the proposed system; an *objectives report*, which defines the goals and objectives of the proposed system; and a *software requirements specification*, which defines the software architecture of the system. The *system survey* consists of the analysis and design diagrams for the current system, a data dictionary, and a list of current system assumptions and limitations. The *objectives report* consist of a prioritized list of objectives, a list of logical requirements for the proposed model, and the top-level data flow diagrams for a proposed solution. (These may have been developed during the feasibility analysis.) The *requirements specification* consists of the analysis diagrams for the new system, a data dictionary, and process specifications. The relationship of these various products are summarized in Table 2, shown below.

System Survey of Current System
- Data Flow Diagrams with Associated Data Dictionary
- Entity Relationship Diagrams with Object Definitions and Relationship Descriptions
- State Transition Diagrams
- Assumptions and Limitations

Objectives Report for Proposed System
- Prioritized List of Objectives
- List of Logical Requirements
- Top Level Data Flow Diagrams for Proposed Solution

Requirements Specification for Proposed System
- Data Flow Diagrams with Associated Data Dictionary
- Entity Relationship Diagrams with Object Definitions and Relationship Descriptions
- State Transition Diagrams
- Process Specifications

Table 2 Form of the Products of Analysis

The outputs of the analysis process have a significant graphics dependency and there are well-defined relationships between these graphical system depictions and the textual descriptions. The proposed tool must support the creation, storage, recall, and modification of the three common graphic diagrams in current popular use for structured analysis. These are Data Flow Diagrams, Entity Relationship Diagrams, and State Transition Diagrams. The automation of the generation of these graphical representations of a software system is crucial to improving the productivity of a programming staff. A sample data flow diagram is shown in Figure 2.

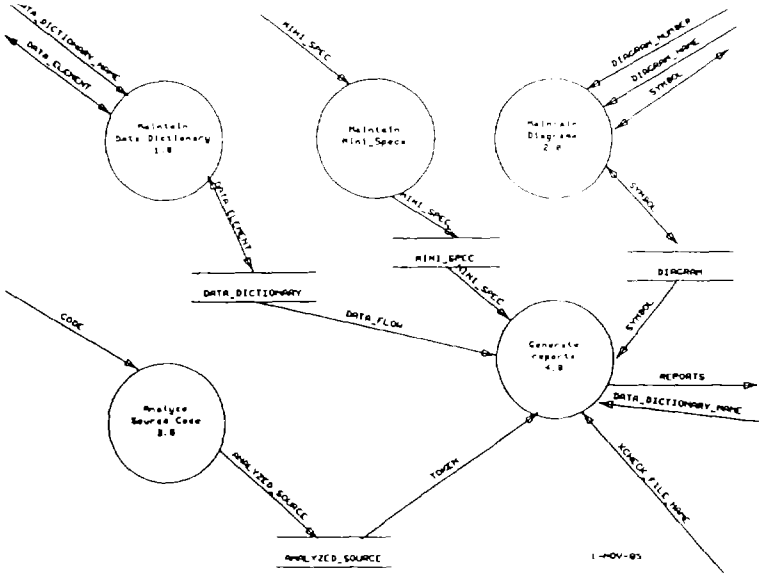


Figure 2 A Dataflow Diagram

This figure illustrates the symbols of the data flow diagram; the circle for a process, two parallel lines for a data store, a single-headed vector for a uni-directional data flow, a two-headed vector for a bi-directional data flow. The Data Flow Diagram graphically shows the flow of data through a system by capturing the processes of a system and their interfaces to the other processes and to the outside world. The dataflows are labelled with meaningful data names that are in the data dictionary. The tool must support fairly long names and readibility is enhanced if the lettering can be sloped as shown. The data flows and data stores on a data flow diagram must be defined in the associated data dictionary. The tool must crosscheck the data dictionary for all data which appears on the data flow diagram. Logical links must be maintained between parent and child diagrams and balancing between levels through the data dictionary must be maintained.

Figure 3 illustrates the symbols of an entity relationship diagram. The rectangle represents an object, a diamond represents a relationship, and connectors are labeled with the cardinality between objects. An Entity Relationship Diagram models the data structures of a system by providing a snapshot of the data and relationships at any point in time. All objects and relationships are described in a dictionary which allows entry of attributes, keys, how the object is used and by whom the object is used. The entity relationship diagram and data flow diagrams are linked through objects and data stores respectively. The tool should perform this consistency check.

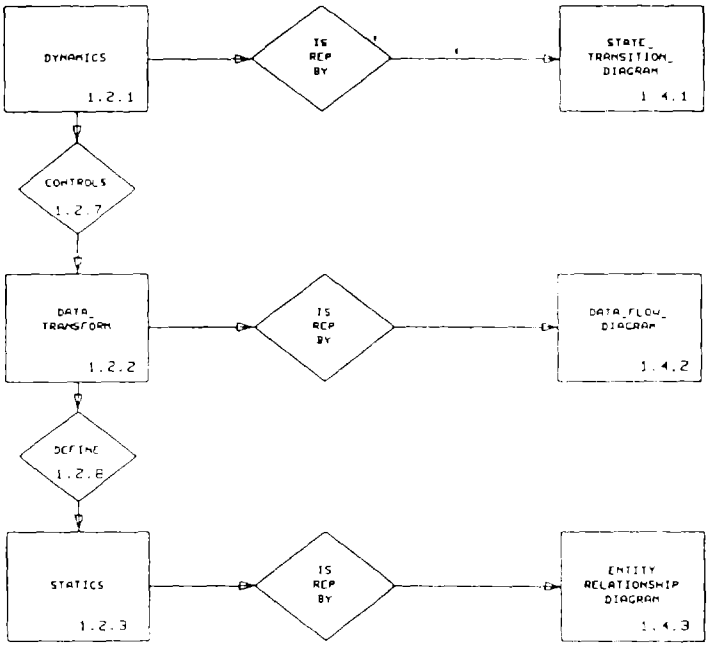


Figure 3 An Entity Relationship Diagram

A State Transition Diagram, as illustrated in Figure 4, models the dynamics of a system. Each rectangle represents a unique state of the system. The vectors show the allowable transition paths between states. A horizontal line separates the conditions causing the transition (text above the line) from the actions taken as a result of the transition (text below the line). State transition diagrams are logically linked to data flow diagrams through the conditions and actions in the transitions to the control flows on the data flow diagrams. Since conditions often reflect a change in

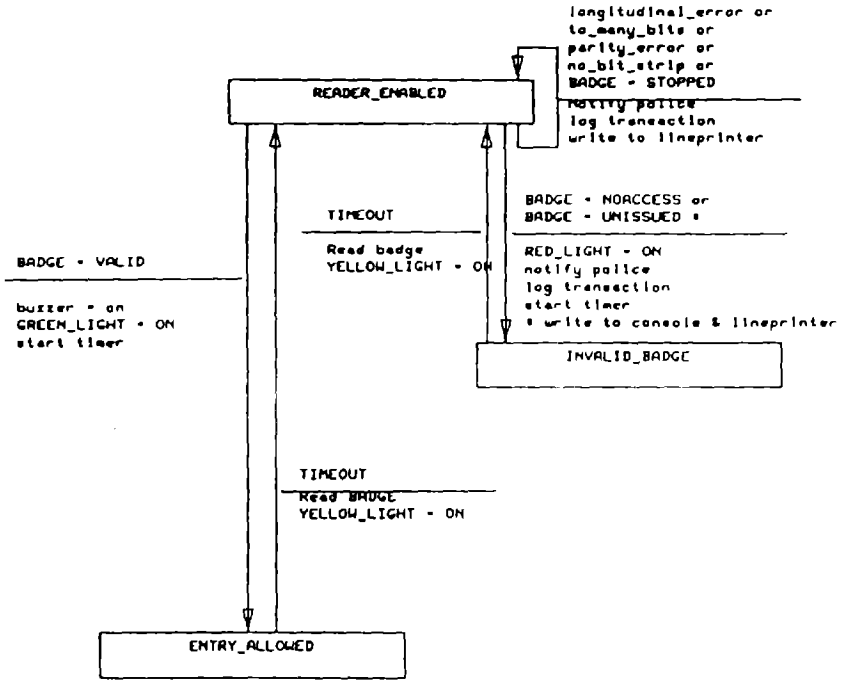


Figure 4 A State Transition Diagram

the value of some piece of data, the tool must confirm that the data is defined in the data dictionary and is consistent with the usage of the data in any other parts of the system specification. Also, since actions caused by a state transition could cause the initiation of a process, any reference to a process in a state transition diagram must be checked for consistency with the description on a data flow diagram.

System Design

The output of the design process is the *design specification*, which consists of module structure diagrams, module specifications, a data dictionary, report layouts, and physical file descriptions, a *module test plan*, a *system test plan*, and a *user guide*. These are summarized below in Table 3.

Design Specification
- Structure Diagrams
- Module Specifications
- Data Dictionary
- Report Layouts
- Physical File Descriptions

Module Test Plan

System Test Plan

User Guide

Table 3 Form of the Products of Design

Structure Diagrams (Figure 5) provide a graphical blueprint for the code. Each box represents a module, vectors point to submodules called by another module or submodule. Data storage is shown by appropriate symbols, and arguments passed are represented by labeled vectors.

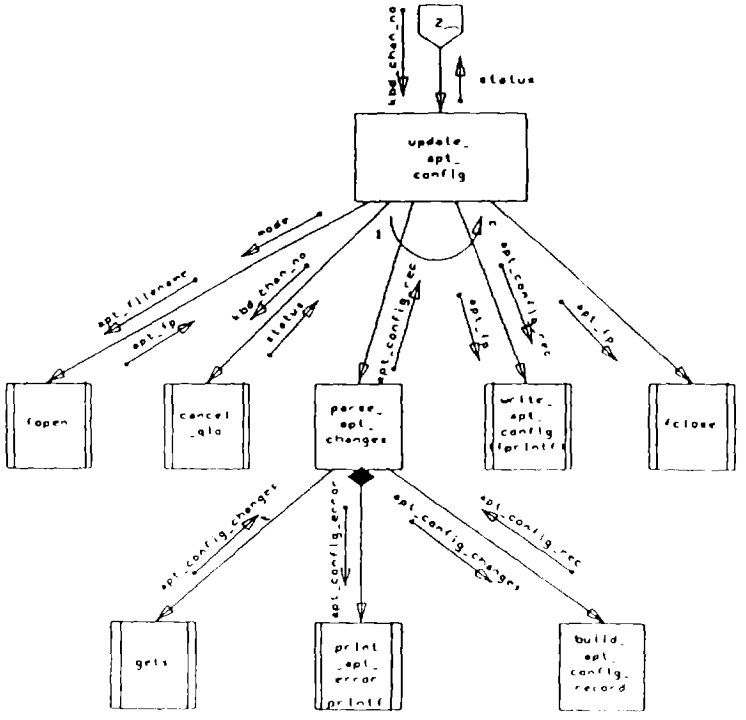


Figure 5 A Module Structure Diagram

Module Specifications use a simple, unambiguous vocabulary for pseudo-code. Variables correspond with those on the Structure Diagrams and are all defined in a data dictionary.

The tool should support the generation of the graphic symbols used in the structure diagram. It should also check for consistency between the data dictionary and the module specifications. All modules shown on the structure diagram should have module specifications which have consistent data requirements. The advanced document processor tool should have standard templates for test plan preparation and user guide generation. Consistency between the integrated system test plan and the original specification should be checked.

Code Modules

Code modules are directly related to the design specification. Structure diagrams and pseudo-code must provide a template for the code. The tool must confirm that the code, the pseudo-code and the structure diagrams are all consistent with each other. A modification to any one of the three must cause an update to the others, automatically. The tool must allow for automatic header generation, suitable to the requirements of the project, to be inserted in each coded module. The tool must link the coded module to the test procedure used to verify the module.

Test Modules

Test beds are developed based on the individual modules to ensure integrity of inputs and outputs, decision branching, and allowable values of variables as defined in the data dictionary. A top-down approach reduces the amount of time needed to test modules. The test plan generated will begin with top level modules and test inputs and outputs, branching and variable bounds. A tool must ensure all modules are thoroughly tested. Tests must be reproduceable to allow for retesting at each new release.

Integrate Modules

Integration Testing confirms that the integrated system meets the requirements as determined in the System Specification. The tool should insure that there is a one-to-one correlation between each requirement and each test case. This ensures completeness of the final product. Consistency of integration test scenarios and respective requirements as detailed in the System Specification should be maintained by the tool.

General User Considerations

Since the proposed tool will be so heavily utilized during the software development process, the user interface is very important. A variety of different interfaces should be supported for the graphic input used for pointing, such as a graph tablet, mouse, or light pen. High user responsiveness (1-3 seconds) should be a goal as should high system performance in general. Hard copy outputs should be available for any screen display and the screen displays should be individually tailorable. All diagrams should be annotatable, both with text and figures. The tool should run on a variety of standardly available computers and workstations, and should also have an interface to a variety of different display terminals. Terminals with limited capability should still have access to the tools set, although with a reduced capability. All internal tool data structures should be transferrable to standard ASCII format for transportability and archiving.

Project Management

From a project managers' point of view, the partitioning of the work can be derived from the data flow diagrams or the module structure diagrams. Since each element of these diagrams has a relationship with a work breakdown structure chart, the software engineering tools should have the capability to tie these logical partitions together. Other management charts, such as Gantt and Pert charts, are also related to the software product and should be maintained by the tool. Task assignments and estimated completion dates should be developed and maintained using the tool. Schedular information (such as a critical path analysis) can be tied to the software diagrams which should be automatically updated as changes are made.

Code Analysis

One of the recurring problems in software development is the maintenance of programs that were developed without a formal record of analysis and design. In many cases, the only documentation that exists for a software system is the source code itself. A software tools package should have a code analyzer capability which would analyze source code and generate structure diagrams and module test templates. A software tools package should have the provision of allowing later analysis and design enhancements to the system to be added to the project database without a complete description of the entire system.

Configuration Management

An integral part of any software engineering tools package is its ability to perform the configuration management and control functions critical to the success of any project. A software engineering tools package should support these functions by maintaining a project database which will maintain, for each element of the project, factors such as access control, development control, and version release control. Access control determines which team members have element modification privileges to a given element, which team members have read-only access, and which team members have no access at all (such as might be encountered in a need-to-know environment). Development control manages the review/walkthrough/release cycle of each element, tracking its status through the development process, allowing release of an element only after design review, approval, coding, and testing. Version release control is needed for projects where a variety of versions or configurations must be re-installable at any particular any point in time.

Extensibility

The package should not merely conform to one vendor's concept of the symbols and methodology of Software Engineering - it should be both flexible enough to support exceptions to 'standard' rules and extensible enough to allow the user to add particular modifications and extensions as desired.

References

1.) Tom DeMarco, "Structured Analysis and System Specification", Yourdon Press, 1979.

2.) Edward Yourdon and Larry L. Constantine, "Structured Design", Yourdon Press, 1978.

3.) Matt Flavin, "Fundamental Concepts of Information Modeling", Yourdon Press, 1981.